

CPE

Apontadores e Alocação Dinâmica de Memória

Departamento de Engenharia Elétrica – UnB

- 1 Endereços de variáveis e apontadores
- 2 Aplicações de apontadores
- 3 Alocação dinâmica de memória

A memória e o programa de computador

- A memória disponível para um programa é dividida em três partes.
 - 1 área do **código** executável do programa;
 - 2 área da **pilha** - área que armazena variáveis locais e parâmetros de funções;
 - 3 área de **dados** - bem maior que a pilha e que serve para alocar variáveis que não couberam e/ou são muito grandes, além de servir para alocar novos espaços pelo programador, durante a **execução**.
- Este tópico ensinará como definir e manipular dinamicamente esses espaços em memória.

...
Código
Pilha
Dados
...

Variáveis

Ao declararmos uma variável `x` como a abaixo:

```
int x = 100;
```

Temos associados a ela os seguintes elementos:

- Um tipo (`int`);
- Um nome (`x`);
- Um endereço de memória ou referência (`0xbfd2`);
- Um valor (`100`).

Na memória

0xbfd2	...
	100
	...

O operador *address-of* (&)

Para acessarmos o endereço de uma variável, usamos o operador & ao lado do seu nome:

Exemplo

```
int x = 100;  
cout << "Valor de x = " << x << endl;  
cout << "Endereco de x = " << &x << endl;
```

Veja o exemplo em `x.cpp`.

Apontador

- Existem tipos de dados para armazenar os **endereços** de variáveis.
- Uma variável declarada como um destes tipos é chamada de **apontador** ou **ponteiro**.
- Ao atribuir o endereço de uma variável a um apontador, dizemos que o mesmo **aponta** para a variável.

Exemplo

```
int x;  
int *ap_x; //apontador para inteiros  
  
ap_x = &x; /* ap_x aponta para x */
```

Vea o exemplo em `ap_x.cpp`.

Apontador

- Se escrevemos os comandos abaixo dentro do nosso programa:

```
int x = 100;
```

```
int *ap_x;
```

```
ap_x = &x;
```

- Na memória temos (endereços escolhidos arbitrariamente):

	...
(x) 0xbfd2	100
	...
(ap_x) 0xbff3	0xbfd2
	...

Declaração de apontadores em C++

Para declarar um apontador/ponteiro utilizamos o operador unário `*`.

Exemplo

```
int    *ap_int;  
char   *ap_char;  
float  *ap_float;  
double *ap_double;
```

Cuidado ao declarar vários apontadores em uma única linha. O operador `*` deve preceder o nome de **cada** variável apontadora.

Veja os exemplos em `apontadores.cpp` e `ap_tabela.cpp`.

Fazendo acesso aos valores das variáveis referenciadas

- Um endereço de variável por si só não é muito útil. Para acessarmos o valor de uma variável apontada por um endereço, também usamos o operador `*`:
 - Ao precedermos um apontador com este operador, obtemos o equivalente a variável armazenada no endereço em questão.
 - `*ap_x` pode ser usado em **qualquer** contexto que a variável `x` seria.

Exemplo

```
int x, *ap_x;  
ap_x = &x;  
*ap_x = 3;
```

Veja o exemplo em `valores.cpp`.

Passagem de parâmetros por valor e referência

- Como já vimos, ao passarmos argumentos para uma função, a princípio estes são copiados como variáveis locais da função. Isto é chamado passagem por valor.
- Já vimos também uma forma de passagem por referência, usando o operador & ao lado do tipo do parâmetro na declaração.

Exemplo

```
void troca(int& x, int& y) {  
    int aux;  
    //troca de valores entre variaveis  
    aux = x;  
    x = y;  
    y = aux;  
}
```

Passagem de parâmetros por valor e referência

- Há uma segunda forma de realizar passagem de argumentos por referência, que consiste em passarmos como argumento o **endereço** da variável, e não o seu valor.
- Ou seja, o mecanismo corresponde a passarmos **apontadores** para as variáveis que queremos alterar na função.

Retomando: passagem de argumentos por referência

- Para indicarmos que será passado o endereço do argumento, usamos o mesmo tipo que usamos para declarar um ponteiro:

```
tipo nome (tipo *parâmetro1, tipo *parâmetro2, ...,  
          tipo *parâmetroN) {  
    comandos;  
}
```

Passagem de argumentos por referência

- Para acessarmos o conteúdo da variável apontada pelo ponteiro, usamos o operador *

```
void troca(int *end_x, int *end_y) {  
    int aux;  
    aux = *end_x;  
    *end_x = *end_y;  
    *end_y = aux;  
}
```

Exemplo: troca_pointer.cpp

```
1  #include <iostream>
2  using namespace std;
3
4  void troca(int *end_x, int *end_y) {
5      int aux;
6      //troca de valores entre var.
7      aux = *end_x;
8      *end_x = *end_y;
9      *end_y = aux;
10 }
11
12 int main() {
13     int a = 100, b = 200;
14
15     troca(&a, &b);
16     cout << "a = " << a << ", b = " << b << endl;
17     return 0;
18 }
```

Repare que, ao chamar a função troca, aí é necessário passar os **endereços** de a e b.

Apontadores e vetores

- A variável que representa um vetor é, por definição, um apontador **constante** para o primeiro elemento do vetor.
- A operação de indexação corresponde a deslocar este apontador ao longo dos elementos alocados ao vetor.
- Isto pode ser feito de duas formas:
 - Usando o operador de indexação ($v[4]$).
 - Usando aritmética de endereços ($*(v+4)$).

Apontadores e vetores

- Esta dupla identidade entre ponteiros e vetores é a responsável pelo fato de vetores serem sempre passados por referência e pela inabilidade da linguagem em detectar acessos fora dos limites de um vetor.

Veja o exemplos em `ap_e_vetor.cpp` e `cadeias.cpp`.

Vetores de apontadores

- Não existe diferença entre vetores de apontadores e vetores de tipos simples.
- Neste caso, basta observar que o operador `*` tem precedência menor que o operador de indexação `[]`.

Exemplo

```
int *vet_ap[5];  
char *vet_cadeias[5];  
  
cout << *vet_ap[0] << ' ' << vet_cadeias[0]);
```

Veja o exemplo em `vet_cadeias.cpp`.

Alocação dinâmica de memória

- Além de reservarmos espaços de memória com tamanho fixo na forma de variáveis locais, podemos reservar espaços de memória de tamanho arbitrário e acessá-los através de apontadores.
- Desta forma podemos escrever programas mais flexíveis, pois nem todos os tamanhos devem ser definidos aos escrever o programa.
 - Por exemplo, a memória necessária pode depender da entrada dada pelo usuário.
- A alocação e liberação destes espaços na área de dados (heap) da memória é feito por meio de dois operadores em C++:
 - `new`: Aloca um espaço de memória.
 - `delete`: Libera um espaço de memória.

Operador new

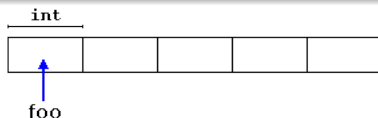
```
ponteiro = new tipo;  
ponteiro = new tipo[numero_de_elementos];
```

- O operador aloca a memória necessária e retorna um **ponteiro** para o início do bloco.

Operador new

Exemplo: um vetor de 5 inteiros

```
int *foo;  
foo = new int[5];
```



- O sistema aloca espaço para 5 elementos do tipo `int` e retorna um ponteiro para o primeiro elemento da sequência, que é atribuído a `foo`.
- Como já visto, o primeiro elemento apontado por `foo` daí pode ser acessado com a expressão `foo[0]` ou `*foo`.
- O segundo elemento pode ser acessado com a expressão `foo[1]` ou `*(foo+1)`, e assim por diante...

Outro exemplo - operador new

Analise o exemplo do arquivo `new.cpp` e o esquema abaixo, que ilustra a situação na memória ao executá-lo (endereços escolhidos arbitrariamente):

	...
(p) 0xafd2	0xbf00
	...
(*p ou p[0]) 0xbf00	0
*(p+1) ou p[1] 0xbf01	1
*(p+2) ou p[2] 0xbf02	2
	...
*(p+99) ou p[99] 0xbf63	99
	...

Operador delete

Uma vez que não seja mais necessário, podemos liberar o uso de um bloco de memória, permitindo que este espaço seja reaproveitado depois ou por outro programa:

```
delete ponteiro;  
delete[] ponteiro;
```

- O primeiro comando é para liberar espaço de um único elemento alocado com o comando `new`, enquanto o segundo comando libera espaço alocado para vetores
- Deve ser passado para o comando `delete` exatamente o mesmo ponteiro retornado pelo operador `new`.

Operador delete

Exemplo: alocando e liberando um vetor de 5 inteiros

```
int *p;  
p = new int[5];  
delete[] p;
```

Veja o exemplo em `delete.cpp`.

Exemplo: alocando e liberando memória de matrizes

- No caso de matrizes ou vetores multidimensionais, devemos alocar um vetor de apontadores (um apontador por linha) e depois um vetor de elementos para cada linha.
- Na hora de liberar o espaço, temos também que aplicar a operação delete para cada linha.

Veja o exemplo em `transpostadinamico.cpp`.

Exemplo final: notasdinamico.cpp

```
1  #include <iostream>
2  #include <cmath>
3  #include <iomanip>
4  using namespace std;
5
6  void leia_notas(double nota[], int n){
7      for (int i = 0; i < n; i++) { //leitura das notas
8          cout << "Nota do aluno " << i+1 << ": ";
9          cin >> nota[i];
10     }
11 }
12 //calcula a media do vetor v
13 double media(double v[], int n){
14     double m = 0.0;
15     for (int i = 0; i < n; i++) //calcula da media
16         m += v[i];
17     m /= n;
18     return m;
19 }
20 //calcula desv. padrao do vetor v de media med
21 double desvpad(double v[], int n, double med){
22     double dv = 0.0;
23     for (int i = 0; i < n; i++)
24         dv += (v[i]-med)*(v[i]-med); //calcula do desvio padrao
25     dv = sqrt(dv/n);
26     return dv;
27 }
```

Continuação: notasdinamico.cpp

```

1  int main() {
2      double *notas, mu, sigma;
3      int tamanho;
4
5      cout << "Numero de alunos: ";
6      cin >> tamanho;
7
8      notas = new double[tamanho]; //alocacao dinamica do vetor
9      leia_notas(notas, tamanho); //chama procedimento que le em notas
10
11     mu = media(notas, tamanho); //chama funcao que calcula media
12     sigma = desvpad(notas, tamanho, mu); //chama funcao que calcula desv. padrao
13
14     delete [] notas; //liberacao da memoria usada pelo vetor;
15     cout << "Nota media = "
16           << fixed
17           << setprecision(2)
18           << mu
19           << "\nDesvio padrao = "
20           << sigma
21           << endl;
22
23     return 0;
24 }

```

Dilbert

